Technical Debt Prioritization: State of the Art. A Systematic Literature Review

The Journal of Systems & Software 171 (2021) 110827

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

A systematic literature review on Technical Debt prioritization: Strategies, processes, factors, and tools

Valentina Lenarduzzi [a,*], Terese Besker [b], Davide Taibi [c], Antonio Martini [d], Francesca Arcelli Fontana [e]

[a] LUT University, Lathi, Finland
[b] Chalmers University of Technology, Göteborg, Sweden
[c] Tampere University, Tampere, Finland
[d] University of Oslo, Oslo, Norway
[e] University of Milano-Bicocca, Milan, Italy

ARTICLE INFO

Article history:
Received 4 February 2020
Received in revised form 22 July 2020
Accepted 14 September 2020
Available online 14 October 2020

Keywords:
Technical Debt
Technical Debt prioritization
Systematic Literature Review

ABSTRACT

Background Software companies need to manage and refactor Technical Debt issues. Therefore, it is necessary to understand if and when refactoring of Technical Debt should be prioritized with respect to developing features or fixing bugs.
Objective The goal of this study is to investigate the existing body of knowledge in software engineering to understand what Technical Debt prioritization approaches have been proposed in research and industry.
Method We conducted a Systematic Literature Review of 557 unique papers published until 2020, following a consolidated methodology applied in software engineering. We included 44 primary studies.
Results Different approaches have been proposed for Technical Debt prioritization, all having different goals and proposing optimization regarding different criteria. The proposed measures capture only a small part of the plethora of factors used to prioritize Technical Debt qualitatively in practice. We present an impact map of such factors. However, there is a lack of empirical and validated set of tools.
Conclusion We observed that Technical Debt prioritization research is preliminary and there is no consensus on what the important factors are and how to measure them. Consequently, we cannot consider current research conclusive. In this paper, we therefore outline different directions for necessary future investigations.

1. Introduction

Technical Debt (TD) is a metaphor introduced by Cunningham (1992) to represent sub-optimal design or implementation solutions that yield a benefit in the short term but make changes more costly or even impossible in the medium to long term (Avgeriou et al., 2016b). Software companies need to manage such sub-optimal solutions. The presence of TD is inevitable (Martini et al., 2015) and even desirable under some circumstances (Besker et al., 2018c) for a number of reasons, which may often be related to unpredictable business or environmental forces internal or external to the organization.

However, just like any other financial debt, every TD has an interest attached, or else an extra cost or negative impact that is generated by the presence of a sub-optimal solution (Li et al., 2015). When such interest becomes very costly, it can lead to disruptive events, such as development crises (Martini et al., 2015). The current best practices employed by software companies include keeping TD at bay by avoiding it if the consequences are known or refactoring or rewriting code and other artifacts in order to get rid of the accumulated sub-optimal solutions and their negative impact. Companies cannot afford to avoid or repay all the TD that is generated continuously and may be unknown (Martini et al., 2015). The main business goals of companies are to continuously deliver value to their customers and to maintain their products.

Thus, the activity of refactoring TD usually competes with the allocation of time to spend on developing new features and fixing defects, where TD refactoring activities are often down prioritized over implementation of new features (Martini et al., 2015). It is therefore of utmost importance to understand when refactoring TD becomes more important than postponing a feature or a

* Corresponding author.
E-mail addresses: valentina.lenarduzzi@lut.fi (V. Lenarduzzi), besker@chalmers.se (T. Besker), davide.taibi@tuni.fi (D. Taibi), antonima@ifi.uio.no (A. Martini), francesca.arcelli@unimib.it (F. Arcelli Fontana).

https://doi.org/10.1016/j.jss.2020.110827

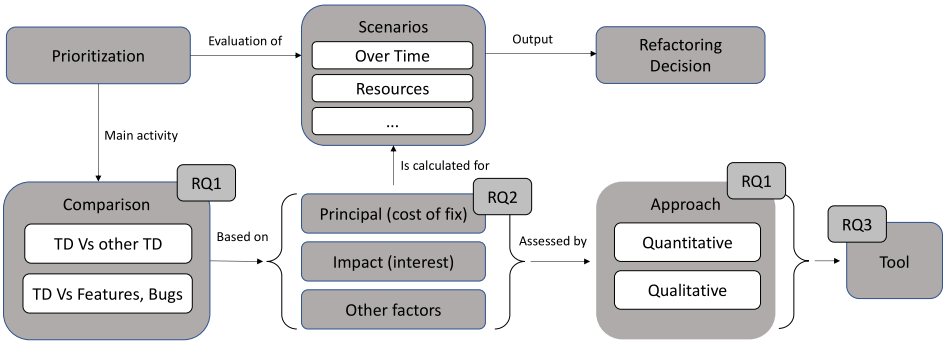

Fig. 1. The TD prioritization mind map.

bug fix. In other words, it is important to understand how to *prioritize TD with respect to features and bugs*. However, a recent study (Besker et al., 2019) performed in several software companies reported how companies struggle to prioritize TD and they do not use systematic approaches. This leads to our first research question: *Which prioritization strategies have been proposed?* (RQ_1) Answering this RQ will provide a list of prioritization strategies that have been studied, which can be useful for practitioners to improve their practices and identify the strategy more suited to their aims.

In addition, recent studies show how different projects and even different types of TD might be associated with different refactoring costs (principal) and negative impact (interest) (Besker et al., 2018b). But it is still not clear in academia and in industry how to measure principal and interest and what tools can help in those activities. We then formulate other RQs: *Which factors and measures have been considered for TD prioritization?* (RQ_2) and *Which tools have been used to prioritize TD?* (RQ_3)

Once again, answering these questions would provide practitioners with actionable measures and tools to quantify or at least estimate principal and interest, and therefore to prioritize TD.

Furthermore, as part of the prioritization strategies related to RQ_1, we know that some TD_s can be more dangerous than others (Seaman et al., 2012; Martini and Bosch, 2016c), and it is therefore important to understand how to *prioritize TD with respect to other TD*. We therefore propose the following RQ, to understand what approaches are available and which ones are instead missing: *Are papers prioritizing TD vs. TD or TD vs. Features?* ($RQ_{1.1}$) Yet another important aspect related to the strategies to prioritize TD, refers to its periodicity, where the prioritization for instance either can be done as a one-shot activity but it can also be done iteratively and being a part of a continuous process.

We therefore propose the RQ: *Is the prioritization based on a one-shot activity or on a continuous process?* ($RQ_{1.2}$). This research question focuses on how the prioritization process is described in the reviewed publications in terms of its periodicity. We distinguish the different approaches in terms of one-shot activities versus part of a continuous process

However, there is no overall study that investigate strategies, processes, factors, and tools for TD prioritization. The only similar work (Alfayez et al., 2020) analyzed prioritization approaches based only on their accounts for value and cost.

Our goal in this paper is to survey the existing body of knowledge in software engineering to understand which approaches have been proposed in research and industry to *prioritize* TD. For this reason, we performed a Systematic Literature Review (SLR) on the prioritization of TD. We conducted a SLR in order to investigate the existing body of knowledge in software engineering to understand how TD is prioritized in software organizations and which research approaches have been proposed.

The main contribution of this paper is a report on the state of the art concerning approaches, factors, measures, and tools used in practice or proposed in research to prioritize TD.

The paper is structured as follows: In Section 3, we describe the background of this review. In Section 4, we outline the research methodology adopted in this study. Sections 5 and 6 present and discuss the obtained results. Finally, in Section 7, we identify the threats to validity and in Section 8 draw the conclusion.

2. The TD prioritization mind map

We propose a preliminary mind map to conceptualize our goal and research questions, that can help practitioners during TD prioritization activities as illustrated in Fig. 1. This Mind Map offers an exploration of different factors that need to be taken into consideration during the TD prioritization process and how these factors can be relates to each other.

The first step is to decide whether the practitioners require a prioritization of the refactorings among TD issues or whether they need to prioritize a TD refactoring versus the implementation of new features and bug fixes (to answer $\mathbf{RQ_1}$). This is because the approaches differ in terms of assessing the impact of TD and assessing the value or the impact of features and bugs. In the former case, the prioritization approach can use the same factors, while in the latter case, it is more probable that the principal and interest of TD need to be compared with feature-oriented factors, for example competitive advantage or cost of delay.

Once the scope of the comparison is defined, the evaluation of TD should be performed taking into account: (1) the difference in the TD principal (the cost of fixing the issues), (2) the impact (the TD interest), and (3) other factors, including economic and marketing factors (results from $\mathbf{RQ_2}$).

The evaluation can be both quantitative and qualitative ($\mathbf{RQ_1}$), and in some cases could be supported by tools ($\mathbf{RQ_3}$). As an example, companies might quantitatively evaluate the presence of Code Debt using tools, but they might also need to perform a qualitative evaluation (e.g., with code reviews) of factors that cannot be measured with tools, for example considering code readability, analyzability, or other quality characteristics. In addition, some tools provide means to calculate the principal of the TD, but practitioners might need to calculate the interest by qualitatively assessing the impact factors.

Moreover, the evaluation should be performed considering different scenarios, including the available resources and the possible evolution of the system. In fact, TD can be quite context-dependent (as we discussed for the impact factors in $\mathbf{RQ_3}$), which means that practitioners need to assess it with estimations of future scenarios. For example, in the tool AnaConDebt, practitioners can specify events happening in short-, medium- or long-term scenarios. The evaluation of the different scenarios

Table 1
Definition of technical Debt (Li et al., 2015).

TD type	Definition
Requirements TD	"Refers to the distance between the optimal requirements specification and the actual system implementation, under domain assumptions and constraints"
Architectural TD	"Is caused by architecture decisions that make compromises in some internal quality aspects, such as maintainability"
Design TD	"Refers to technical shortcuts that are taken in detailed design"
Code TD	"Is the poorly written code that violates best coding practices or coding rules. Examples include code duplication and over- complex code"
Test TD	"Refers to shortcuts taken in testing. An example is lack of tests (e.g., unit tests, integration tests, and acceptance tests)"
Build TD	"Refers to flaws in a software system, in its build system, or in its build process that make the build overly complex and difficult"
Documentation TD	"Refers to insufficient, incomplete, or outdated documentation in any aspect of software development. Examples include out-of-date architecture documentation and lack of code comments"
Infrastructure TD	"Refers to a sub-optimal configuration of development-related processes, technologies, supporting tools, etc. Such a sub-optimal configuration negatively affects the team's ability to produce a quality product"
Versioning TD	"Refers to the problems in source code versioning, such as unnecessary code forks"
Defect TD	"Refers to defects, bugs, or failures found in software systems"

Table 2
Previous SLR_s.

ID	Year	Goal
Tom et al. (2013)	2012	Understanding the nature of TD
Li et al. (2015)	2015	TD management and TD classification
Ampatzoglou et al. (2015)	2015	Financial approaches for managing TD
Ribeiro et al. (2016)	2016	TD payment prioritization
Alves et al. (2016)	2016	TD management strategies, TD taxonomy
Fernández-Sánchez et al. (2017)	2017	TD management elements
Behutiye et al. (2017)	2017	TD in Agile development
Besker et al. (2018a)	2018	Managing architectural TD
Rios et al. (2018)	2018	TD types, management strategies
Khomyakov et al. (2019)	2019	TD tools
Alfayez et al. (2020)	2020	TD effort

should help in making refactoring decisions, for example regarding which refactorings should be performed and which should be postponed.

As an example of the decision process, a company might consider not implementing a new feature that involves a code section or module that is suffering from TD. This can happen if such TD is estimated to generate high interest in the short-term scenario: In such a case, the interest generated by the TD could overcome the cost of delaying the feature. The practitioners might then decide to refactor the code before implementing the feature.

Let us take a concrete example of how a refactoring decision is made following the steps in the prioritization mind map. An architect needs to decide whether to refactor a "sub-optimal" interface before more applications accessing it are developed. The main activity of the architect is to evaluate whether to prioritize the refactoring of TD vs. developing new features. Then the architect needs to take into consideration and calculate different factors (principal, interest, and other factors). Without the refactoring, the TD would spread to all the new code (*Contagious debt*). In addition, all the new applications would suffer from the negative impact (interest) generated by interacting with the sub-optimal API (*Spread of impact in the system*. Although delaying the development of the new applications (feature-oriented factor) would imply costs in the short-term scenario, the lead time for developing new features in the long-term scenario could be reduced as the developers would not pay the interest generated by the sub-optimal interface. If such long-term gain overcomes the cost of delaying the application development, the practitioners should choose to perform the refactoring of the API. In this case, the refactoring decision would be made by evaluating whether, in a future scenario, the cost of avoiding the interest is worth paying the principal.

The TD prioritization Mind Map can assist practitioners, in combination with the other results presented in this paper (impact map, description of prioritization approaches, and available tools), in reaching a refactoring decision.

3. Background

In this Section, we explain the meaning of TD in order to avoid confusion or misunderstandings, and we will report on previously published systematic reviews.

3.1. Technical Debt

The concept of TD was introduced for the first time in 1992 by Cunningham as *"The debt incurred through the speeding up of software project development which results in a number of deficiencies ending up in high maintenance overheads"* (Cunningham, 1992). In 2013, McConnell (2013) refined the definition of TD as *"A design or construction approach that is expedient in the short term but that creates a technical context in which the same work will cost more to do later than it would cost to do now (including increased cost over time)"*. In 2016, Avgeriou et al. (2016a) defined it as *"A collection of design or implementation constructs that are expedient in the short term, but set up a technical context that can make future changes more costly or impossible. TD presents an actual or contingent liability whose impact is limited to internal system qualities, primarily maintainability and evolvability"*.

Li et al. (2015) conducted a systematic mapping study for understanding the concept of TD and created an overview of the current state of research on managing TD. Based on the selected studies (96), they proposed a classification of ten types of TD at different levels, as reported in Table 1. Since this classification derives from a recent secondary study and is, according to our knowledge, the most complete one available in the literature, we considered it in our search strategy process (Section 4.2) to define our search terms.

3.2. Previous SLR_s

In this Section, we briefly report on previous systematic reviews (Systematic Mapping Studies and Systematic Literature Reviews) available in the source engines, showing their main

goals in Table 2). We present the studies in chronological order in order to show the research evolution regarding TD. The first systematic review was published in 2012 (Tom et al., 2013) and the last ones, to the best of our knowledge, in 2018 (Besker et al., 2018a; Rios et al., 2018).

Tom et al. (2013) exploited an exploratory case study technique that involves a multivocal literature review, supplemented by interviews with software practitioners and academics, in order to establish the boundaries of the TD phenomenon. As a result, they created a theoretical framework that provides a holistic view of TD, comprising a set of TD dimensions, attributes, precedents, and outcomes. The framework provides a useful approach to understanding the overall phenomenon of TD for practical purposes.

Li et al. (2015) investigated TD management (TDM), providing a classification of TD concepts and presenting the current state of research on TDM. They considered publications between 1992 and 2013, ultimately selecting 94 studies. The results showed a need for empirical studies with high-quality evidence on the TDM process, application of TDM approaches in industrial contexts, and tools for managing the different TD types during the TDM process.

Ampatzoglou et al. (2015) analyzed research efforts regarding TD, focusing on financial aspects underlying software engineering concepts. They considered publications until 2015, selecting 69 studies. The results provide a glossary of terms and a classification scheme for financial approaches to be applied for managing TD. Moreover, they discovered that a clear mapping between financial and software engineering concepts is lacking.

Ribeiro et al. (2016) evaluated the appropriate time for paying a TD item and how to apply decision-making criteria to balance the short-term benefits against long-term costs. They considered publications until 2016, selecting 38 studies. They identified 14 decision-making criteria that can be used by development teams to prioritize the payment of TD items and a list of types of debt related to the criteria.

Alves et al. (2016) investigated what strategies have been proposed to identify and manage TD in software projects, considering publications between 2010 and 2014 and selecting 100 studies. They proposed an initial taxonomy of TD types and provided a list of indicators to identify TD and management strategies. Moreover, they analyzed the current state on TD, highlighting possible research gaps. The results showed a growing interest of researchers in the TD area. They identified some gaps regarding new indicator proposals and management strategies and tools for controlling TD. Another gap they identified regards empirical studies for validating the proposed strategies.

Fernández-Sánchez et al. (2017) identified the elements needed to manage TD, considering publications until 2017 and selecting 69 studies. They did not provide a general overview of the TD phenomenon or of the activities for managing TD. The elements were classified into three groups (basic decision-making factors, cost estimation techniques, practices and techniques for decision-making) and grouped based on stakeholders' points of view (engineering, engineering management, and business-organizational management).

Behutiye et al. (2017) analyzed the state of the art of TD and its causes, consequences, and management strategies in the context of agile software development (ASD). They considered publications until 2017 and selected 38 studies, finding potential research areas for further investigation. The study highlighted positive interest in TD and ASD and provided some potential categories that can easily lead to TD, such as "focus on quick delivery and architectural and design issues".

Besker et al. (2018a) investigated Architectural TD (ATD), synthesizing and compiling research efforts in order to create new knowledge with a specific interest in ATD. They considered publications between 2005 and 2016, selecting 43 studies. The results showed a lack of guidelines on how to manage ATD successfully in practice and of an overall process where these activities are fully integrated.

Rios et al. (2018) performed a tertiary study based on a set of five research questions and evaluated 13 secondary studies dating from 2012 to March 2018. They evolved a taxonomy of TD types, identified a list of situations in which debt items can be found in software projects, and organized a map representing the state of the art of activities, strategies, and tools for supporting TD management. Their results can help to identify points that still require further investigation in TD research. For example, they found that there are management activities that do not have any type of support tool.

Khomyakov et al. (2019) investigated existing tools for the measurement and analysis of TD, focusing on quantitative methods that could also be automated. They selected 21 papers out of 331 retrieved. Their results show that many new approaches are being defined to measure TD.

Avgeriou et al. (2020) compared the existing tools for measuring TD, comparing their features and popularity, and analyzing the existing empirical evidence on their validity. Differently from this work, they did not compare the methods adopted to prioritize TD.

Recently, Alfayez et al. (2020) investigated the software artifact dependencies, and type of required human involvement. They analyzed prioritization approaches based on their accounts for value, cost, or resources constraints.

Comparing to the existing SLR_s, our study is the only one that investigates strategies, processes, factors, and tools for TD prioritization considering all the possible aspects and not focusing only on a specific ones (such as effort Ribeiro et al., 2016; Alfayez et al., 2020, or development process Behutiye et al., 2017).

4. Methodology

In order to understand the state of the art and the practice on Technical Debt prioritization, we conducted a systematic literature review based on the guidelines defined by Kitchenham and Charters (2007) and Kitchenham and Brereton (2013). We also applied the "snowballing" process defined by Wohlin (2014).

In this Section, we describe the goal and the research questions (Section 4.1) and report our search strategy approach (Section 4.2). Moreover, we performed a quality assessment (Section 4.3) for each included paper and outlined the data extraction and the analysis (Section 4.4) of the corresponding data.

4.1. Goal and research questions

Based on our mind map, the study goal was to investigate the existing body of knowledge in software engineering to understand how TD is prioritized in software organizations and what research approaches have been proposed.

Based on our goal, we defined the following research questions (**RQ_s**):

RQ_1	Which prioritization strategies have been proposed?
$RQ_{1.1}$	Are papers prioritizing TD vs. TD or TD vs. Features?
$RQ_{1.2}$	Is the prioritization based on a one-shot activity or on a continuous process?
RQ_2	Which factors and measures have been considered for TD prioritization?
RQ_3	Which tools have been used to prioritize TD?

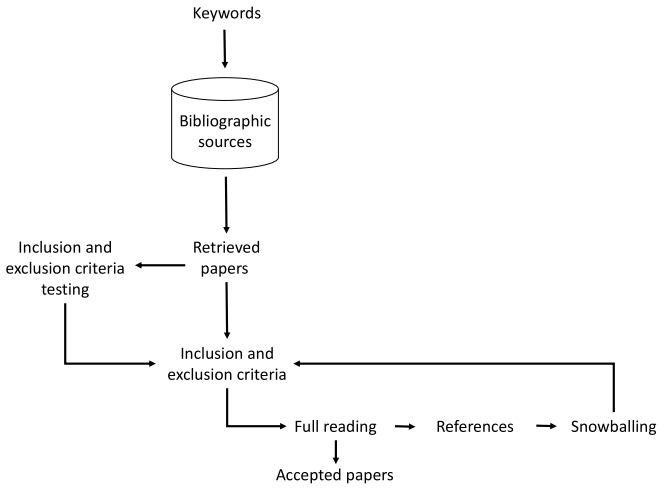

Fig. 2. The search and selection process.

The first research question targets how the investigated research papers address the prioritization process of TD, both in terms of different strategies (**RQ**$_1$), i.e., whether the prioritization process of TD mainly focuses on different TD items or also includes prioritization between TD items and, e.g., the implementation of new features (**RQ**$_{1.1}$), and of how the prioritization process is described in terms of its periodicity (**RQ**$_{1.2}$).

Based on the above RQ$_s$, we aimed at identifying a set of factors and measures considered useful during TD prioritization activities (**RQ**$_2$). Moreover, we aimed at understanding which measures are considered in the prioritization of the main TD components, principal and interest. We aim to provide a list of existing tools used to evaluate TD in order to depict the current situation in terms of numbers and the maturity of each tool (**RQ**$_3$).

4.2. Search strategy

The search strategy involves the outline of the most relevant bibliographic sources and search terms, the definition of the inclusion and exclusion criteria, and the selection process relevant for the inclusion decision. Our search strategy is depicted in Fig. 2.

Search terms. In our search string, we included all the terms related to TD proposed by Li et al. (2015) and reported in Table 1 (Section 3).

The search string contained the following search terms:

("technical debt")OR ("design debt") OR ("architect* debt") OR ("test* debt") OR ("implem* debt") OR ("docum* debt") OR ("requirement debt") OR ("code debt") OR ("Infrastructure debt") OR ("versioning debt") OR ("defect debt") OR ("build debt")

We used the asterisk character (*) for the second term group in order to capture possible term variations such as plurals and verb conjugations. To increase the likelihood of finding publications addressing TD prioritization, we applied the search string to both title and abstract.

Bibliographic sources. We selected the list of relevant bibliographic sources following the suggestions of Kitchenham and Charters (2007), since these sources are recognized as the most representative in the software engineering domain and used in many reviews. The list includes: *ACM Digital Library, IEEEXplore Digital Library, Science Direct, Scopus, Google Scholar, CiteSeer library, Inspec, Springer link*. Moreover, we performed a manual search on the most important conferences and workshops on Technical Debt, such as the International Conference on Technical Debt (TechDebt).

Table 3
Inclusion and exclusion criteria.

Criteria	Assessment criteria	Step
Inclusion	Papers that prioritize TD issues	All
	Papers that report the criteria of removal, refactoring, remediation of TD issues regarding any aspect (financial, maintenance, performance, readability, …)	All
	Papers that compare TD issues	All
	Papers that empirically investigated the prioritization of removal, refactoring, remediation of TD issues	F
Exclusion	Papers not fully written in English	T/A
	Papers not peer-reviewed (i.e., blog, forum ...)	
	Duplicate papers (only consider the most recent version)	T/A
	Position papers and work plans (i.e., papers that do not report results)	T/A
	Publications where the full paper cannot be located (i.e., if database used does not have access to the full text of the publication)	T/A
	Publications that only mention prioritization of TD in an introductory statement and do not fully or partly focus on it	All
	Only the latest version of the papers (e.g., journal papers that extend conference papers are excluded if they refer to the same dataset)	All

Inclusion and exclusion criteria. We defined inclusion and exclusion criteria to be applied to the title and abstract (T/A) or to the full text (F) or to both cases (All), as reported in Table 3.

Search and selection process. The search was conducted in December 2019 and included all the publications available until this period. The application of the searching terms returned 557 unique papers.

Testing the applicability of inclusion and exclusion criteria: Before applying the inclusion and exclusion criteria, we tested their applicability (Kitchenham and Brereton, 2013) on a subset of ten papers (assigned to all the authors) randomly selected from the papers retrieved.

Applying inclusion and exclusion criteria to title and abstract: We applied the refined criteria to the remaining 547 papers. Each paper was read by two authors; in the case of disagreement, a third author was involved in the discussion to clear up any such disagreement. For 29 papers, we involved a third author. Out of the 557 initial papers, we included 116 based on title and abstract.

Full reading: We fully read the 117 papers included by title and abstract, applying the criteria defined in Table 3 and assigning each one to two authors. We involved a third author for six papers to reach a final decision. Based on this step, we selected 49 papers as possibly relevant contributions.

Snowballing: We performed the snowballing process (Wohlin, 2014), considering all the references presented in the retrieved papers and evaluating all the papers referencing the retrieved ones, which resulted in one additional relevant paper. We applied the same process as for the retrieved papers. The snowballing search was conducted in December 2019. We identified only 11 potential papers, but only one of these was included in order to compose the final set of publications.

Based on the search and selection process, we retrieved a total of 50 papers for the review, as reported in Table 5.

4.3. Quality assessment

Before proceeding with the review, we checked whether the quality of the selected papers was sufficient to support our goal

Table 4
Quality assessment criteria.

QA_s	Quality Assessment Criteria (QA)	Response scale
QA_1	Is the paper based on research (or is it merely a "lessons learned" report based on expert opinion)?	
QA_2	Is there a clear statement of the aims of the research?	
QA_3	Is there an adequate description of the context in which the research was carried out?	
QA_4	Was the research design appropriate to address the aims of the research?	Excellent = 4
QA_5	Was the recruitment strategy appropriate for the aims of the research?	Very Good=3
QA_6	Was there a control group with which to compare treatments?	Good=2
QA_7	Was the data collected in a way that addressed the research issue?	Fair=1
QA_8	Was the data analysis sufficiently rigorous?	Poor=0
QA_9	Has the relationship between researcher and participants been considered to an adequate degree?	
QA_{10}	Is there a clear statement of findings?	
QA_{11}	Is the study of value for research or practice?	

Table 5
Results of search and selection and application of quality assessment criteria.

Step	# Papers
Retrieval from bibliographic sources (unique papers)	557
Reading by title and abstract	439 rejected
Full reading	68 rejected
Backward and forward snowballing	1
Papers identified	50
Quality assessment	6 rejected
Primary studies	**44**

and whether the quality of each paper reached a certain quality level. We performed this step according to the protocol proposed by Dybå and Dingsøyr (2008). To evaluate the selected papers, we prepared a checklist (Table 4) with a set of specific questions. We ranked each answer, assigning a score on a five-point Likert scale (0 = poor, 4 = excellent). A paper satisfied the quality assessment criteria if it achieved a rating higher than (or equal to) 2.

Among the 50 papers included in the review from the search and selection process, only 44 fulfilled the quality assessment criteria, as reported in Table 5.

4.4. Data extraction

We extracted data from the 44 primary studies (PSs) that satisfied the quality assessment criteria. The data extraction form, together with the mapping of the information needed to answer to each RQs, is summarized in Table 6.

In order to answer **RQ**$_1$, we first extracted data related to the context of the papers, outlining each PS in terms of the type of TD evaluated, according to the list proposed by Li et al. (2015). We also reported the type of evaluation adopted in the papers, distinguishing between qualitative, quantitative, and mixed evaluation approaches. Moreover, we also extracted the criteria of removal, refactoring or remediation of TD issues.

We extracted data related to the goal of the prioritization (**RQ**$_{1.1}$), so as to understand if papers prioritized TD vs. TD or TD vs. the implementation of new features. To answer **RQ**$_{1.2}$, we reported if the prioritization is based on a one-shot activity, or on a continuous process, if it is proactive or reactive.

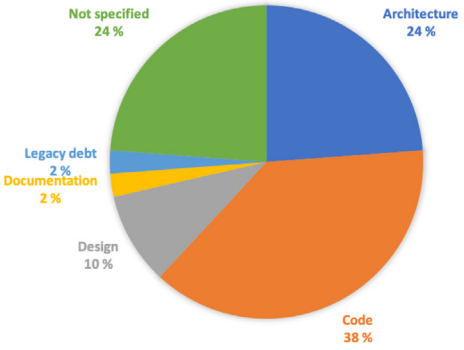

Fig. 3. Types of TD.

To answer **RQ**$_2$, we extracted the measures and factors used to assess the prioritization of a TD issue and which of these are suggested during the prioritization process.

Finally, to answer **RQ**$_3$, we retrieved information about the frameworks and tools adopted to evaluate and prioritize TD issues. This data is exclusively based on what is reported in the papers, without any kind of personal interpretation.

4.5. Replicability

In order to allow replication and extension of our work by other researchers, we prepared a replication package[1] for this study with the complete results obtained.

5. Results

Based on the adopted selection process, we identified 44 primary studies (PS_s). Considering the TD type reported in Table 1, the types of TD considered most frequently in the PS_s were: Code Debt (38%), Architectural Debt (24%), and Design Debt (10%). Moreover, some PS_s (24%) do not report on issues of any specific TD type, but evaluate TD in general (Fig. 3).

Code TD is generally investigated from the point of view of its impact on one – or more than one – software qualities [SP13], [SP18], [SP19], [SP26]. Maintainability [SP4], [SP5], [SP11] and maintenance effort [SP1], [SP2], [SP11], [SP19] are considered most often by the PS_s. Code debt evaluation is mostly based on code smells [SP2], [SP5], [SP11], [SP18], [SP19], [SP26].

Other metrics are also considered, such as the time [SP4], [SP23] or cost [SP1] needed to fix a violation, and quality rules [SP13].

Some factors related to subjective evaluation such as customer feedback [SP23] or developers' comments in the code [SP28] are evaluated less often.

The approaches mainly involve models that reduce TD by removing or refactoring code smells or other metrics [SP11],[SP18]. These approaches look at the impact on code smells [SP5], make a comparison with classes without smells [SP2], [SP26], or rank the code rules [SP13] perceived as critical by developers.

Architectural TD is general investigated taking into account the role of architectural smells [SP17], [SP19], [SP20] or complex architectural design [SP17], [SP27] which negatively impact software quality [SP17], [SP19], [SP20]. Architectural TD is evaluated by measuring the extra maintenance effort for bug fixing [SP17] or analyzing the bug-proneness [SP17] of the code. Another approach combines three different perspectives, such as historical

[1] http://www.taibi.it/raw-data/JSS_TD_2019.zip (The raw data will be moved to a permanent repository (Mendeley Data) in case of acceptance of this paper).

Table 6
Data extraction.

RQ_s	Data	Outcome
RQ_1	TD type	Architectural, Build, Code, Defect, Design, Documentation, Infrastructure, Requirement, Test, and Versioning
RQ_1	Evaluation type	Qualitative, quantitative, or mixed evaluation approach
RQ_1	Criteria of removal, refactoring, remediation of TD issues	
$RQ_{1.1}$	Prioritization goal	TD vs. TD, TD vs. Feature
$RQ_{1.2}$	Process type	Single activity or continuous process, proactive or reactive
RQ_2	Measures and factors used to assess the prioritization of a TD issue	e.g. a specific code smell, or metric, or feature
RQ_3	Frameworks and tools adopted	

data of the projects, architectural design, and severity of the class prioritizing the refactoring activities [SP19].

Architectural design is used to identify high interest in terms of wasted time related to architectural TD [SP27], combined with other metrics such as number of files and percentage of complex functions and files [SP35].

Another approach identifies dependencies and social gaps across architecture organization in order to define architectural TD [SP31].

5.1. RQ_1 which prioritization strategies have been proposed?

TD prioritization is considered as one of the most important activities when managing TD. The TD prioritization process is used for defining the ordering and/or scheduling of planned refactoring initiatives based on the priority of each identified TD item concerning the impact of the individual items on the software. Several different prioritization aspects have been proposed by researchers in the reviewed publications and a few methods on how to prioritize TD have been developed, but there is no unified approach regarding how the TD prioritization process should be carried out, nor is there a consensus on which aspects to focus on when performing the TD prioritization process. The selection of the prioritization strategy is currently context-dependent in most organizations [SP21].

In order to analyze the prioritization aspects presented in the retrieved publications, a thematic analysis approach was used. Thematic analysis is an effective method for identifying, analyzing, and reporting patterns and themes within a searched data scope (Braun and Clarke, 2006). The thematic analysis returned mainly five themes illustrating different prioritization aspects. However, one should note that from a software evolution perspective, these aspects can potentially have dependencies and couplings.

Based on the analysis, the different suggested prioritization strategies presented in the reviewed publications are mainly: (a) improving software quality, (b) increasing software practitioners' productivity, (c) affection on the correctness of the software, (d) cost–benefit analysis (CBA) to compare various TD items with respect to low cost and high payoff, or (e) a combination of several different approaches.

These different strategies are presented in Fig. 4 together with references to the publications representing each type of strategy.

Following sub-sections will describe more in detail how each of the investigated publications contribute to the illustrated categorization of TD strategies.

5.1.1. Internal software quality

Studies focusing on internal software quality as a prioritization strategy commonly focus on a quality assessment of the software in order to identify the TD items that cause the highest maintenance costs [SP1], [SP2], [SP13], [SP19], [SP28], [SP26], [SP4], [SP31], [SP35], [SP41], [SP44], [SP3], [SP9], [SP17], [SP42], [SP40] together with factors such as remaining product life, debt severity and its impact on future development activities, and current business-related constraints [SP3], [SP9].

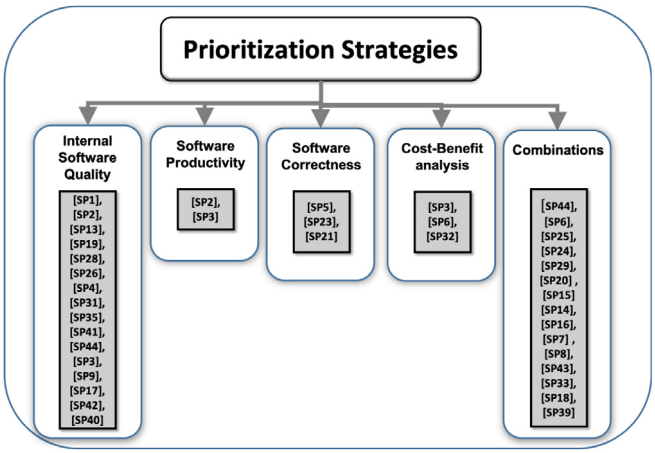

Fig. 4. Technical debt prioritization strategies.

Xiao et al. [SP17] suggest an approach that focuses on architectural TD. It focuses both on locating TD items and on ranking and prioritizing them. Their approach returns the TD items that consume the largest maintenance effort and therefore deserve more attention and higher priority for refactoring

Plösh et al. proposes a TD prioritization approach with a primarily focus on the prioritization of Design debt, and their approach relies on the quantification of design best practices by transferring the identified TD items into a portfolio-matrix [SP42]. Albarak and Bahsoon further claim that software systems having database tables below fourth normal form are likely to form TD and therefore the ill-normalized tables should be prioritized for refactoring [SP40].

5.1.2. Software productivity

Some reviewed publications also take the decrease in software practitioners' productivity into consideration when prioritizing TD, since software suffering from architectural TD, for example, slows down development by causing rework [SP2], [SP3].

5.1.3. Software correctness

The effect TD has on the correctness of the software is described as an approach for evaluating different candidate TD items for prioritization [SP2]. More specifically, Arcelli Fontana, Ferme and Spinelli [SP5] report that the prioritization of the refactoring of code smells representing design debt can be evaluated by studying the impact of the refactoring of the code smells on

different quality metrics, with the goal is to identify and prioritize "the most dangerous smell and hence the smell which represents the worst TD". When prioritizing defect debt, in particular, Akbarinasaji et al. [SP23] focus their approach on the severity of the debt items (using the categorizations critical, major, normal, and minor) and the duration of bug-xing time.

Codabux et al. [SP21] used a Bayesian approach to build a prediction model for determining the "TD proneness" of each TD item using a classification scheme according to the TD proneness probability where the risk of the individual items is assessed.

5.1.4. Cost–benefit analysis

Researchers such as [SP3], [SP6] use a cost–benefit analysis when prioritizing different TD items, focusing on which refactoring activities should be performed first because they are likely to be inexpensive to implement yet have a significant effect, and which refactoring should be postponed due to high cost and low payoff. The main focus of this approach is on making a lucrative investment in the software, with the output of this analysis being a prioritized list of different TD items ordered by the profitability of the different possible refactoring activities [SP3].

This strategy is echoed by Martini et al. [SP32], who state that "if the interest is (or is going to be) high, the debt is worth being paid. On the contrary, if the interest is not enough to justify the cost of refactoring, there is no reason to "waste resources to refactor the system". However, Martini et al. [SP32] also stress the importance of not only focusing the prioritization decisions on single TD items by assessing each TD item separately, but also understanding the overall impact TD items generally have on the whole project, thus focusing on the overall project goals by evaluating the information holistically. In this approach, Martini et al. [SP32] also include factors such as the portion of the code affected by the TD, the project size, the roadmap, the positive impact of the TD, the existence of an alternative, and the cultural attitude of the team when prioritizing TD refactoring activities.

5.1.5. A combination of several different approaches

Quite a few of the investigated publication suggest a combination of several different approaches. For instance, Alfayez and Boehm [SP44] propose an automated search-based approach for prioritizing TD using a multi-objective evolutionary algorithm called MOEA (which is an open-source Java library), having a focus on the repayment of the TD refactoring activity within a specific cost constraint.

Borrowing prioritization approaches from other disciplines, such as finance and psychology, Seaman et al. [SP6] include techniques such as Analytic Hierarchy Process (AHP), the Portfolio method, and the Options approach. The AHP approach involves building a criteria hierarchy, assigning weights and scales to the criteria, and finally performing a series of pairwise comparisons between the alternatives against the various criteria. The goal of using the Portfolio approach is to select those assets that maximize the return on investment or minimize the investment risk.

Codabux et al. [SP25] stress the importance of adopting a broader perspective on the prioritization process, focusing on the liability of TD. According to them, decision makers need to think beyond the cost associated with fixing the debt, including estimates of the possible future costs resulting from the decision to ship. The additional costs reflected during the prioritization in terms of liability costs include, e.g., costs for responding to support requests, costs associated with catastrophic failures, etc., and potential litigation costs if service level agreements are violated because of unmanageable debts.

Ribeiro et al. [SP24] present a multiple decision strategy criteria model using a combination of different prioritization approaches, which can be used during different project phases. Their model focuses on aspects such as, e.g., the severity of the impact the TD items have from a customer perspective on the interest cost of TD, the lifetime of the project's properties, and its possibility of evolution.

Yet another prioritization process that includes different perspectives is the approach described by Ciolkowski et al. [SP29]. Their approach focuses on a combination of the overall software quality with a focus on productivity improvement from a future-oriented perspective, using a proactive methodology.

Gupta et al. [SP20] use a two-level approach for prioritizing TD. First, the TD items are assessed according to their importance and urgency. In a second step, the TD items' impact on business values and effort is assessed.

Guo et al. [SP15] present a TD prioritization approach that ranks customer expectations according to top priority, followed by availability of development resources, the interest of the TD items, the current status of the debt-infected modules, and the impact of the debt on other features. By studying how software practitioners prioritize TD items in practice, Yli-Huumo et al. [SP14], [SP16] concluded that their prioritization approach commonly focuses on scalability, business value, use of a feature, and customer effect.

Snipes et al. [SP7] suggest a TD prioritization approach that includes a combination of factors such as severity, the existence of a workaround, urgency of the refactoring required by customers, refactoring effort, the risk of the proposed refactoring, and the scope of testing required.

Schmid [SP8] distinguishes between potential and effective TD, where potential TD is any type of suboptimal software system, while effective TD refers to issues in the software system that make further development of that system more difficult. This prioritization approach considers aspects such as evolution cost, refactoring cost, and the probability that the predicted evolution path will be realized.

Further, Almeida et al. [SP43] suggest to also focus on business objectives when prioritization TD in order to support business expectations and goals. The researcher compared the differences between a technical prioritization and a business-oriented one, and they state that their results show that "taking business priorities into account can change decisions related to technical debt prioritization". This prioritization aspect is also described to facilitate the argumentation from the technical side and thereby to convince business stakeholders to prioritize what was previously considered pure-technical problems.

Martini and Bosch [SP33] propose a tool called AnaConDebt to provide assistance during the TD prioritization process. Their tool assesses the severity of the interest for different TD items, with the calculation of the interest being based on an assessment of seven different factors and their growth. The assessed factors are: (1) reduced development speed, (2) bugs related to the TD item, (3) other qualities compromised, (4) other extra costs, (5) frequency of the issue, (6) spread in the system, and (7) users affected. Vidal et al. [SP18] also propose a tool called JSpIRIT for specifically prioritizing source-code-related TD, where the TD items are evaluated according to their importance based on different prioritization criteria. The tool calculates a ranking for a set of code smells according to their importance, where the tool can instantiate to prioritize TD items by different criteria. Examples of such criteria are the relevance of the kind of code smells, the history of the system, or different software metrics, among others. Additionally, the developer can use external information to improve the prioritization.

Yet another reviewed publication [SP39] suggests performing TD prioritization using a tool called CodeScene, where factors such as how developers work with the code is taken into consideration. The process uses an complexity trend analysis when calculating the indentation-based complexity of the identified TD items and together with a skilled human observer set out the final TD prioritization.

5.2. $RQ_{1.1}$ are papers prioritizing TD vs. TD or TD vs. features?

Since today's software companies face increasing pressure to deliver customer value, the balance between spending developer time, effort, and resources on implementing new features or spending it on TD remediation activities, on fixing bugs, or on other system improvements become vital. In this study, we limited the scope to studying the balance between prioritizing the implementation of new features or the remediation of existing TD.

To conclude, this research question seeks to address whether the TD prioritization process mainly focuses on the prioritization among different TD items or whether the TD items are described as competing with the implementation of new features or not.

Budget, resources, and available time are important factors in a software project, especially during the prioritization process, since spending time and effort on refactoring activities commonly infers that less time can be spent on implementing new features, for example. This is one of the main reasons why software companies do not always spend additional budget and effort on the refactoring of TD since they commonly have a strong focus on delivering customer-visible features [SP18].

Ciolkowski et al. [SP29] describe this situation like this: "The challenge for project managers is to nd a balance when using the given budget and schedule, either by reducing TD or by adding technical features. This balance is needed to keep time to market for current product releases short and future maintenance costs at an acceptable level". Echo this view stating that "Ideally, actionable refactoring targets should be prioritized based on the technical debt interest rate to balance the trade-os between improvements, risk, and new features" [SP39].

Furthermore, Martini, Bosch and Chaudron [SP10] state that TD refactoring initiatives usually get low priority compared to the implementation of new features and that TD that is not directly related to the implementation of new features is often postponed.

Vathsavayi and Systa [SP22] echo this notion, stating that "Deciding whether to spend resources for developing new features or fixing the debt is a challenging task". The researchers highlight that software teams need to prioritize new features, bug fixes, and TD refactoring within the same prioritization process.

However, even if the balance between implementing new features and TD refactoring activities is described as important [SP31], the papers investigated in this study commonly focus their prioritization approaches on prioritization among different TD items, with the goal being to determine which item should be refactored first. None of the prioritization approaches described in the surveyed publications explicitly addresses how the prioritization between implementing new features and spending time and effort on the refactoring of TD should be carried out. However, the study by Besker et al. (2019) states that "the pressure of delivering customer value and meeting delivery deadlines forces the software teams to down-prioritize TD refactorings continuously in favor of implementing new features rapidly".

5.3. $RQ_{1.2}$ is the prioritization based on a one-shot activity or on a continuous process?

Just as important as prioritizing TD refactoring activities in a project is to describe a management strategy for the prioritization process.

Therefore, this research question focuses on how the prioritization process is described in the reviewed publications in terms of its periodicity. We distinguish the different approaches in terms of one-shot activities versus part of a continuous process.

Some of the publications reviewed in this study highlight the TD prioritization process in terms of it being a continuous, integrated, and iterative process [SP16], [SP22], whereas others stress the importance of prioritizing TD refactoring within each sprint [SP15]. Choudhary et al. [SP19] illustrate the prioritization process as being an integral part of the continuous development process by saying "ideally software companies try to incorporate refactoring practices as an integral part of their development and maintenance processes" [SP9], and [SP39] echos this notion stating that "a systematic management of TD and how to reduce it should also be considered important in each release of the development project".

Interestingly, however, the rest of the publications reviewed in this study do not give any explicit recommendations on how often or in what way the prioritization of TD should be carried out.

5.4. RQ_2 which factors and measures have been considered for TD prioritization?

When we analyzed the papers to understand which factors were used for TD prioritization, we used a bottom-up approach (inductive analysis). With a deductive analysis (with a priori categories decided in advance), we would have risked to miss some of the factors.

First, we coded each paper with the factors that were mentioned and we obtained a long list of <factor, paper>. Then we grouped and renamed the overlapping factors, to obtain a list of <factor, (paper1, paper2, …)>. Finally, we grouped the factors in categories based on different aspects of software development. The results of this process, especially for the factors related to the interest, are visible in Table 7.

During the prioritization process, six PS_s considered both principal and interest ([SP1], [SP10], [SP13], [SP15], [SP23], [SP35]), while four PS_s considered only interest ([SP13], [SP17], [SP27], [SP34]).

Principal is calculated as cost [SP1], [SP10] or time [SP1], [SP4] needed to fix technical quality issues [SP1] or violations of quality rules [SP13]. Other factors are also considered, such as page rank or customer feedback [SP23].

Interest is calculated as extra cost spent on maintenance due to technical quality issues [SP1], [SP10], [SP17], [SP35] or as wasted time related to different activities (management or refactoring) [SP27], [SP34]. Principal is compared with interest without considering any item for which the benefit does not outweigh the cost [SP15]. The factors considered are: customer expectations, which have the top priority, followed by availability of development resources, the interest of the TD items, the current status of the debt-infected modules, and the impact of the debt on other features [SP15].

In Table 7, we present an "Impact Map", which highlights the plethora of factors related to the impact (interest) of TD to be considered for prioritization, and their wide variation across studies and projects. In total, we counted 53 unique factors.

A few of the factors might overlap, although in different papers the factors are calculated differently. For example, "number of bugs" and "ROI (calculated on number of bugs)" are obviously overlapping factors, although using the sheer number of bugs or the cost of their impact as indicators might give very different results when prioritizing. In other cases, a generic concept of "interest" or "cost" has been used, although such values were probably implicitly calculated by the researchers or practitioners taking in consideration some of the other 52 remaining factors explicitly mentioned in the other papers. However, given the reported information, there is no way to perform such a mapping. Thus, we report a generic factor, for example "risk", as different from all the other specific ones.

The factors have been grouped into categories, when possible, to help navigate them. First, we mapped the factors to qualities

Table 7
Impact Map: Factors and measures related to the interest of TD considered when prioritizing (RQ3).

Category	Factors	PS_s	
		ID	#
Business	Competitive advantage	[SP10]	**3**
	Lead time	[SP10]	
	Attractiveness for the market	[SP10]	
	Penalties	[SP10]	
	Feature usage	[SP16]	
	Business value	[SP16]	
	ROI (calculated per bug)	[SP20]	
Customer	Satisfaction	[SP12]	**5**
	Long-term satisfaction	[SP10]	
	Specific customer value	[SP10]	
	Customer expectations	[SP13]	
	Customer effect	[SP16], [SP24]	
Evolution	Time of impact on evolution (short- or long-term)	[SP8]	**5**
	Risk of critical impact on evolution (possible crisis)	[SP8]	
	Impact on other features	[SP13], [SP24]	
	Impact on upcoming features	[SP22], [SP24], [SP32]	
Maintenance	Modifiability	[SP2], [SP18], [SP26], [SP28]	**12**
	Number of bugs	[SP2], [SP10], [SP11], [SP17], [SP20], [SP23], [SP28], [SP32], [SP33], [SP38]	
	Maintenance cost	[SP10], [SP17], [SP35]	
System qualities	Robustness	[SP4]	**6**
	Performance efficiency	[SP2], [SP4], [SP12]	
	Security	[SP4]	
	Transferability	[SP4]	
	Scalability	[SP16]	
	Generic qualities	[SP32], [SP33], [SP38]	
Quality Debt	# of issues or their co-occurrence	[SP9], [SP16], [SP28], [SP29], [SP25], [SP32], [SP35], [SP36]	**8**
Productivity	% Wasted time (effort)	[SP27], [SP32], [SP33], [SP34], [SP35], [SP38]	**7**
	Number of developers working on TD	[SP35]	
	Wasted development hours	[SP35]	
	Generic effort	[SP24]	
	Coding output/effort	[SP29]	
Project factors	Availability of resources	[SP13]	**3**
	Project size and complexity	[SP32]	
	Postponement of bugs	[SP23]	
Social factors	Developers' morale	[SP30]	**3**
	Social debt	[SP31]	
	Positive impact of TD	[SP32]	
	Team culture	[SP32]	
Other factors	Contagious debt	[SP10]	**6**
	Existence of TD solution (alternative)	[SP32]	
	Spread of impact in the system	[SP32], [SP33], [SP38]	
	Number of users affected	[SP32], [SP33], [SP38]	
	Frequency of negative impact	[SP32], [SP33], [SP38]	
	Kind of smell	[SP18], [SP24]	
	History of the system	[SP18]	
	Compromise architecture	[SP18]	
	Future cost	[SP22]	
	User perception	[SP24]	
Not specified	Risk	[SP10], [SP25]	**8**
	Interest likelihood	[SP13], [SP22]	
	Interest	[SP13], [SP24]	
	Severity	[SP24], [SP38]	
	Customizable	[SP18], [SP24], [SP25], [SP32], [SP33], [SP38]	

that are mentioned most often in relation with TD. These categories are "Evolution", "Maintenance", and "Productivity". For example, the current working definition of TD explicitly mentions the impact on maintainability and evolvability. Given the emphasis on such qualities, we first grouped the factors according to them. TD impacting other qualities was gathered under "System Qualities" (which do not include the former two). Productivity is also usually associated with TD in the form of extra effort spent because of the debt.

Next, we proceeded to categorize and group the remaining factors according to what aspect of software development the impact is related to. This can be important in order to understand which roles would be hit the most by such impact and what consequences it might have on the prioritization. As an example, TD can have a direct impact on the "Customer" factors, so such TD might be considered more important by some organizations in their prioritization. Understanding the impact on "Business" factors can also be very useful in a prioritization against features that are prioritized mostly using business concerns. "Social" and "Project" factors need to be taken into consideration as well, as non-technical aspects of software development.

For some of the factors, it was not possible to find a common category ("Other factors"), or they were only described as high-level factors without additional details ("Not specified").

The majority of the papers focus on the impact of TD on maintainability (12). Some papers focus on productivity (7), evolvability (5), and other system qualities (6), while five papers consider the customer perspective.

Table 8
Tool used when prioritizing TD (RQ4)

Tool name	Tool link	Paper ID
AnaConDebt	https://anacondebt.com	[SP32], [SP33]
ARCAN (Arcelli Fontana et al., 2016; Fontana et al., 2017)	http://essere.disco.unimib.it/wiki/arcan	[SP38]
CAFFEA	Not available	[SP31]
CAST	https://www.castsoftware.com	[SP4]
Coverity	http://www.coverity.com	[SP20]
Findbugs	http://findbugs.sourceforge.net	[SP20]
Visual studio FxCopAnalyzer	https://www.nuget.org/packages/Microsoft.CodeAnalysis.FxCopAnalyzers	[SP20]
iPlasma	http://loose.cs.upt.ro/index.php?n=Main.IPlasma	[SP5]
Jsprit	https://sites.google.com/site/santiagoavidal	[SP18]
Scitool understand	https://scitools.com	[SP21]
SonarQube	https://www.sonarqube.org	[SP30]
Codescene	https://codescene.io	[SP39]

Only a few papers take into consideration other factors, such business factors (3), social factors (3), project factors (3), and other non-categorized factors (6). In most of these cases (including the customer aspect), the identified factors have been reported in a single paper or two. This highlights either their specificity for a specific context or a lack of focus on these factors in the literature. In both [SP10] and [SP24], the authors conducted a survey with practitioners to understand which of these factors are most important for developers, architects, and product owners. In most cases, customer and business factors were considered the most important ones. However, only a few papers address such factors when prioritizing TD, so we can conclude that these factors have been overlooked in the literature.

In quite a few studies (8), the interest (impact) of TD has been identified and assessed as generic interest, interest likelihood, risk, severity, or as customizable by the practitioners. Six papers present factors not categorized specifically in the previously mentioned categories and that represent the impact of TD spanning multiple categories or represent a specific aspect not related to these categories.

Eight other papers assume that the impact of TD is associated with the (co-)occurrence of instances of different issues (e.g., code smells) that are considered sub-optimal ("quantity of debt" in the table). However, the measures used in different papers differ according to the tools used, and the impact of the individual issues is assumed to be the same or was assigned arbitrarily. Very few papers (4) use an estimate or a measure of the cost of refactoring (principal) in contrast to the impact of TD (interest). This is in contrast with the theoretical approach (Chatzigeorgiou et al., 2015; Martini and Bosch, 2016b, [SP8]), according to which TD needs to be prioritized by taking into consideration both the cost of refactoring and the impact.

5.5. RQ_3 which tools have been used to prioritize TD?

As reported in Table 8, only 14 papers mentioned the usage of tools for evaluating and prioritizing TD, but only ten of them report information on which tools were used. The other studies used a custom-made tool developed for their specific purposes.

Out of the aforementioned tools, we can identify ten static analysis tools: *ARCAN*, *CAST*, *Coverity*, *Findbugs*, *Visual Studio FxCopAnalyzer*, *iPlasma*, *Jspirit*, *Scitool Understand*, and *SonarQube*. *Scitool Understand* analyzes the code and visualizes its architecture. The remaining ones detect TD issues such as code or architectural smells, security violations, or others. *CAST*, *Coverity*, *Findbugs*, *Studio FxCopAnalyzer*, *Codescene*, and *SonarQube* are commercial tools commonly used to analyze code compliance against a set of rules. When the rules are violated, they raise a TD issue. These tools provide the severity of the issues and classify them into different types (e.g., issues that could lead to bugs, to increased software maintenance effort, or to security vulnerabilities). Moreover, *CAST* and *SonarQube* also associate a remediation effort (principal), the time needed to remove the TD issue. *ARCAN*, *iPlasma*, and *Jspirit* are open-source tools, developed by research teams and aimed at detecting architectural smells (*ARCAN*) and code smells (*iPlasma* and *Jspirit*).

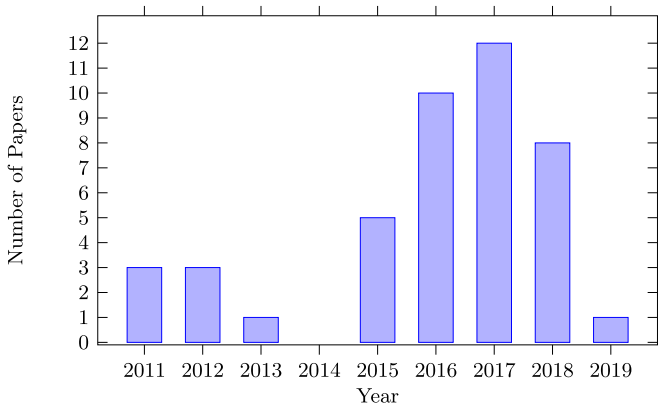

Fig. 5. Paper distribution by year.

AnaConDebt (Martini, 2018) is a management tool based on a TD-enhanced backlog. The backlog allows the creation of TD items and performs TD-specific operations on the created items. In [SP32] and [SP33], *AnaConDebt* has been used to report and visualize the information on TD manually collected by product managers and developers.

The *CAFFEA* framework (Martini and Bosch, 2016a) identifies organizational roles, where architectural responsibilities are allocated. Moreover, the tool defines the team members and share among them. The framework has been used in [SP31] to analyze mismatches between the architecture community and the system architecture.

ARCAN was used in [SP38] to detect architectural smells. The TD principal was then investigated by means of a survey in a large company.

In [SP30], developers were asked to discuss the TD issues raised by *SonarQube*. However, there is no information on whether the developers considered the severity or the type of TD issues. In [SP4], the authors used *CAST* as is to estimate the principal calculated as time to remove all TD issues.

iPlasma and *Jspirit* were used in [SP5] and [SP18], respectively, to detect the number of code smells to be refactored in the systems under investigation.

Scitool Understand was used in [SP21] to identify architectural issues in the system under investigation.

The TD issues detected by *Coverity*, *Findbugs*, and *Visual Studio FxCopAnalyser* were used in [SP20] for an industrial survey.

6. Discussion

In this section, we will discuss the results obtained outlining some implications for researchers and practitioners working in the TD domain. Although the TD domain is relatively young compared to other domains such as software testing or software quality, significant contributions have been published in the last ten years and researchers are becoming more and more active (Fig. 5).

Among the ten TD types proposed in 2015 by Li et al. (2015) (Table 1), only Code Debt and Architectural Debt have been considered frequently by researchers (**RQ**$_1$) in the context of TD prioritization. In the study proposed by Li et al. (2015), Code Debt was the most commonly investigated type of TD, followed by Test Debt. However, other types of TD have also received significant attention. Differently than in Li et al. (2015), in our work it emerged that Code Debt and Architectural Debt are by far the most frequently investigated types of debt when considering TD prioritization. This could be due to the fact that they are easy to measure, mainly based on extensions of previous research from other domains, or it may also be due to the fact that they (particularly ATD) are considered as the most harmful and expensive types to manage in software. For example, architectural and code patterns have been investigated for more than twenty years, even though they were not considered as "debt".

The two most commonly considered types of TD (Code Debt and Architectural Debt) are mainly evaluated by means of architectural or code-level anti-patterns (architectural smells, code smells, or code violations). Moreover, their harmfulness is mainly related to the influence they have on some external quality (e.g., the impact of a specific code smell on maintenance effort). However, their influence is still not clear, since the vast majority of studies do not agree on their harmfulness. Other types of TD should be investigated in the future. We believe that Code Debt is the type investigated most often since it is easy to access the data by mining software repository studies, while other types of debt require other types of studies, including case studies involving developers. We recommend that practitioners should consider the measures identified in this RQ, but should complement them with expert judgment to understand which architectural smells, code smells, or code violations to consider.

In a software affected by TD, the only significantly effective way to reduce this TD is to refactor it. This fact stresses the importance of continuously and iteratively prioritizing the identified refactoring tasks and thereby highlights the importance of using an appropriate TD prioritization process. Through this study, we have identified several different approaches and strategies for prioritizing TD (**RQ**$_1$, **RQ**$_{1.1}$, and **RQ**$_{1.2}$). However, there is no unified approach for this activity, nor there is a consensus on which aspects to focus on when performing the TD prioritization process.

It is evidently clear from the findings that the prioritization process of TD refactoring can be carried out using different approaches, all having different goals and proposing optimization with regard to different criteria.

This study has identified five different main approaches that aim to: (a) improve software qualities, especially maintainability and evolvability, (b) increase software practitioners' productivity, (c) reduce the fault-proneness of the software, (d) compare various TD items using cost–benefit analysis (CBA) to understand the convenience of refactoring, and (e) combine several different approaches.

This result is of value to both academics and practitioners and illustrates that it is important to first identify the goals of TD prioritization, and thereafter to implement a corresponding TD prioritization approach targeting the identified and specified goals.

One interesting finding is that the investigated papers usually only compared different TD items during this prioritizing process and more rarely compared the need for implementing a new feature with the refactoring of TD.

Regarding the characteristics and measures considered during the prioritization process (**RQ**$_2$), the results so far imply that prioritizing TD is an activity that requires a holistic view of several factors. The systematic assessment of TD requires a wide amount of information, which might change from case to case, and in most cases TD is prioritized without following a standardized approach. Also, the known measures used in a few papers capture only a small part of the factors that are used to prioritize TD (proxy for maintenance costs or productivity). Using only such measures to prioritize TD without considering the full picture of the relevant factors (risks and costs) might consequently result in partial and thus biased prioritization, which in turn could lead to poor business decisions. On the other hand, some of the factors have been reported in a single study conducted in a specific context and might not be relevant in other prioritization cases.

More studies are necessary in order to obtain better evidence on factors that have been overlooked (for example factors related to customers, business, social, and project aspects). In addition, we need to better understand which factors should be considered in different contexts, and which additional measures should be considered when prioritizing TD. Finally, although a few holistic approaches have been reported (Martini and Bosch, 2016b, [SP24], [SP33]), there is a need for a better defined framework and a standardized approach for assessing TD.

Considering the two main components of TD, only a limited number of papers propose how to evaluate principal and interest. Interest is mainly calculated as extra cost, or as time wasted to fix TD issues. The reason could be that TD interest is not easy to calculate without access to empirical data from companies. Researchers should design and perform studies to understand the actual interest of existing TD issues.

The tool support for prioritization activities is very fragmented (**RQ**$_3$), which highlights the lack of a solid, widely used, and validated set of tools specifically for TD prioritization. Current tools mainly identify TD issues and, in some cases, propose an estimate of the time needed to fix them. However, to the best of our knowledge, no tools calculate the interest due to the postponement of activities.

Implication for practitioners and researchers

This work highlighted some implications for researchers and practitioners pointing out interesting topics to be investigated in more depth. It might be relevant to understand why researchers focus mainly on code and architectural TD, without considering the other types of TD where TD also seems to be relevant and worthy of investigation (e.g. test or infrastructure TD). Moreover, it might be useful to understand, if the reason is due to the lack of knowledge in primes areas (such as in methods or approaches), limited data, or due to the low levels of interest from the academia. Researchers can also evaluate other important factors such as different approaches, affecting variables, and prioritization of different types of TD. Additionally, not only should researchers investigate the prioritization of different TD items, they should also consider the effect of refactoring TD during the development process (such as the addition of new features introduced into the code) as well as the time and effort spent refactoring the TD. Practitioners may really benefit from these investigations and possibly avoid or reduce the accumulation of TD from the start of the development process.

Moreover, it is important to reach a common consensus regarding how to calculate the interest of TD. Companies can estimate the effort (principal) to fix a particular issue (such as a

code smell or a code style violation), but they cannot estimate the impact and the extra cost for postponing the fixing of the issue. Postponing fixing activities might also have a ripple effect, such as impact other part of the system. As an example, postponing the adoption of a software library might cause developers to write extra code that may not be needed when the library will be finally adopted. The lack of knowledge regarding how to calculate the TD interest may lead to dramatic effects such as increasing costs and decreasing software development quality, which may possibly have a negative impact on the customers. A clear method to calculate TD interest may allow the companies to schedule their refactoring activities based on preferences and avoid the accumulation of TD that might become unmanageable in the future. To solve this issue, researchers need empirical data. In this case practitioners can play a crucial role by taking an active part in the empirical study, providing the data that researchers are missing.

Researchers should also focus their attention to other factors that may lead to TD. Technical aspects are not the only ones that should be considered. Factors related to customers, business, and social are less investigated but might provide another interesting prospective, and possibly open an interdisciplinary research direction.

Since the available tools are not fully mature, research activities can focus on empirical validation of existing tools, confirming the usefulness of each measure proposed by each tool. Practitioners can benefit from our results by using our impact map to explore/anticipate what kind of impact might occur because of TD. Moreover, they should be careful in the selection of the tools, not applying only one but considering more than one.

7. Threats to validity

The results of an SLR may be subject to validity threats, mainly concerning the correctness and completeness of the survey. In this Section, we will outline some implications for researchers and practitioners working in the TD domain. We have structured this Section as proposed by Wohlin et al. (2012), including construct, internal, external, and conclusion validity threats.

7.1. Construct validity

Construct validity is related to generalization of the result to the concept or theory behind the study execution (Wohlin et al., 2012). In our case, it is related to the potentially subjective analysis of the selected studies. As recommended by Kitchenham's guidelines (Kitchenham and Charters, 2007), data extraction was performed independently by two or more researchers and, in case of discrepancies, a third author was involved in the discussion to clear up any disagreement. Moreover, the quality of each selected paper was checked according to the protocol proposed by Dybå and Dingsøyr (2008).

7.2. Internal validity

Internal validity threats are related to possible wrong conclusions about causal relationships between treatment and outcome (Wohlin et al., 2012). In the case of secondary studies, internal validity represents how well the findings represent the findings reported in the literature. In order to address these threats, we carefully followed the tactics proposed by Kitchenham and Charters (2007).

7.3. External validity

External validity threats are related to the ability to generalize the result (Wohlin et al., 2012). In secondary studies, external validity depends on the validity of the selected studies. If the selected studies are not externally valid, the synthesis of its content will not be valid either. In our work, we were not able to evaluate the external validity of all the included studies.

7.4. Conclusion validity

Conclusion validity is related to the reliability of the conclusions drawn from the results (Wohlin et al., 2012). In our case, threats are related to the potential non-inclusion of some studies. In order to mitigate this threat, we carefully applied the search strategy, performing the search in eight digital libraries in conjunction with the snowballing process (Wohlin, 2014), considering all the references presented in the retrieved papers, and evaluating all the papers that reference the retrieved ones, which resulted in one additional relevant paper. We applied a broad search string, which led to a large set of articles, but enabled us to include more possible results. We defined inclusion and exclusion criteria and applied them first to title and abstract. However, we did not rely exclusively on titles and abstracts to establish whether the work reported evidence on Technical Debt prioritization. Before accepting a paper based on title and abstract, we browsed the full text, again applying our inclusion and exclusion criteria.

8. Conclusion

Software companies need to manage and refactor TD issues since sometimes their presence is inevitable, due to a number of causes that may be related to unpredictable business or environmental forces internal or external to the organization. Moreover, some types of TD can be more dangerous than others. Therefore, it is necessary to understand when refactoring TD should be prioritized with respect to implementing features or fixing bugs, or with respect to other types of TD.

We conducted an SLR in order to investigate the existing body of knowledge in software engineering and gain an understanding of how TD is prioritized in software organizations and what research approaches have been proposed. The SLR process was carried out by following two rigorous approaches. We included scientific articles indexed by the most important bibliographic sources and selected by a rigorous process. We considered articles published before December 2019. Our work is based on 38 selected studies, which include data on the state of the art concerning approaches, factors, measures, and tools used in practice or proposed in research to prioritize TD.

The results of our review show that Code Debt and Architectural Debt are by far the most frequently investigated type of debt when considering TD prioritization, while there is scant evidence about other types of TD such as Test Debt and Requirement Debt. The prioritization process of TD refactoring can be carried out using different approaches, all having different goals and proposing optimization with regard to different criteria. However, the identified measures used in a few papers capture only a small part of the factors that are used to prioritize TD.

There is a lack of empirical evidence on measuring principal and interest. Moreover, our results highlight the lack of a solid, validated, and widely used set of tools specifically for TD prioritization.

In practice, we found that there is a plethora of aspects that need to be considered when prioritizing TD. We presented an

impact map of such factors, which can be used as a comprehensive reference regarding which interest might be paid by an organization and how it should be considered. This map can also be used to follow up with further research.

Future work should focus on the investigation of types of TD that have been investigated less often. Moreover, we are planning to investigate how to systematically evaluate and measure the principal and interest of different types of TD. We also aim at developing a framework to support decision-making related to the prioritization of TD.

CRediT authorship contribution statement

Valentina Lenarduzzi: Conceptualization, Methodology, Writing - original draft. **Terese Besker:** Data extraction, Data analysis. **Davide Taibi:** Methodology, Writing - original draft. **Antonio Martini:** Supervision, Writing - review & editing, Funding acquisition. **Francesca Arcelli Fontana:** Supervision, Writing - review & editing, Funding acquisition.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Appendix A. Overview of the primary studies

Based on the adopted selection process, we identified 44 primary studies (PS_s). We illustrate the distribution by year in Fig. 5.

The first three relevant papers on TD prioritization were published in 2011. In the next two years, between 2012 and 2014, only three papers were published. From 2015, the publication trend increased a lot (5 papers), experiencing a considerable increase in 2016, 2017, and 2018 with 10, 12, and 8 papers, respectively.

The selected PS_s are published in 22 different sources, including 6 journals and 15 conferences and workshops. Specifically, the journal publication sources are: (2 papers) Information and Software Technology (IST), (2 papers) Journal of Systems and Software (JSS), (2 papers) IEEE Software, (1 paper) Empirical Software Engineering Journal (EMSE), (1 paper) Journal of Software: Evolution and Process (JSEP), (1 paper) Science of Computer Programming.

Regarding conferences and workshops, the numbers are: (10 papers) International Conference on Technical Debt (TechDebt) (former Workshop on Managing Technical Debt (MTD)), (4 papers) Euromicro Conference on Software Engineering and Advanced Applications (SEAA),(3 papers) International Conference on Agile Software Development (XP), (2 papers) International Conference on Product-Focused Software Process Improvement (PROFES), (2 papers) International Conference on Software Engineering (ICSE), (1 paper) International Conference on Management of Digital Eco Systems (MEDES), (1 paper) International Conference on Services Computing (SCCC), (1 paper) International Workshop on Quantitative Approaches to Software Quality (QuASoQ), (1 paper) International Workshop on Emerging Trends in Software Metrics (WETSoM), (1 paper) International Conference on Enterprise Information Systems (ICEIS), (1 paper) International Symposium on Empirical Software Engineering and Measurement (ESEM), (1 paper) International Conference On Software Architecture Workshop (ICSAW), (1 paper) International Conference on Software Maintenance and Evolution (ICSME), (1 paper) International Conference on Quality of Software architectures (QoSA), (1 paper) International Conference on Software Engineering Advances (ICSEA).

Context Data. 28 PS_s (75.67%) conducted case studies in order to investigate TD issues, analyzing different sets of projects. 24 out 28 PS_s report the findings for each analyzed project in terms of projects number, project size, and programming language.

Regarding the number of projects analyzed, the majority of the PS_s considered fewer than seven each, with most considering only one project. We identified three papers that took into account as context a huge number of projects, such as [SP4] with 700 projects, [SP1] with 44 projects, and [SP5] with 12 projects. Only 11 PS_s report on the programming language of the project(s), with Java, C#, and C++ being the most common ones.

The remaining papers investigated TD issues based on surveys among different practitioners.

TD issues were mainly (48.64%) investigated with a focus on the maintainability process. The remaining PS_s took into account different process phases such as defectively or changeability.

Appendix B. The selected papers (P_s)

[SP1] A. Nugroho, J. Visser, and T. Kuipers. An empirical model of technical debt and interest. 2nd Workshop on Managing Technical Debt (MTD '11). pp. 1–8, 2011.

[SP2] N. Zazworka, M. A. Shaw, F. Shull, and C. Seaman. Investigating the impact of design debt on software quality. 2nd Workshop on Managing Technical Debt (MTD '11). pp. 17–23, 2011.

[SP3] N. Zazworka, C. Seaman, and F. Shull. Prioritizing design debt investment opportunities. 2nd Workshop on Managing Technical Debt (MTD '11). pp. 39–42, 2011.

[SP4] B. Curtis, J. Sappidi and A. Szynkarski. Estimating the Principal of an Application's Technical Debt. IEEE Software, vol. 29, no. 6, pp. 34–42, 2012.

[SP5] F. Arcelli Fontana, V. Ferme, and S. Spinelli. Investigating the impact of code smells debt on quality code evaluation. Third International Workshop on Managing Technical Debt (MTD '12). pp. 15–22, 2012.

[SP6] C. Seaman et al. Using technical debt data in decision making: Potential decision approaches. Third International Workshop on Managing Technical Debt (MTD'12), pp. 45–48, 2012.

[SP7] W. Snipes, B. Robinson, Y. Guo, and C. Seaman. Defining the decision factors for managing defects: a technical debt perspective. Third International Workshop on Managing Technical Debt (MTD '12), pp. 54–60, 2012.

[SP8] K. Schmid. A formal approach to technical debt decision making. 9th international ACM Sigsoft conference on Quality of software architectures (QoSA '13), pp/153-162, 2013.

[SP9] T. Sharma, G. Suryanarayana and G. Samarthyam. Challenges to and Solutions for Refactoring Adoption: An Industrial Perspective. IEEE Software, vol. 32, no. 6, pp. 44–51, 2015.

[SP10] A. Martini, J. Bosch, M. Chaudron. Investigating Architectural Technical Debt accumulation and refactoring over time: A multiple-case study. Information and Software Technology, Volume 67, pp. 237–253, 2015.

[SP11] H. Wang, M. Kessentini, W. Grosky, and H. Meddeb. On the use of time series and search based software engineering for refactoring recommendation. 7th International Conference on Management of computational and collective intElligence in Digital EcoSystems (MEDES '15). pp. 35–42, 2015.

[SP12] A. Martini and J. Bosch. Towards Prioritizing Architecture Technical Debt: Information Needs of Architects and Product Owners. 41st Euromicro Conference on Software Engineering and Advanced Applications. pp. 422–429, 2015.

[SP13] D. Falessi and A. Voegele. Validating and prioritizing quality rules for managing technical debt: An industrial case study. 7th International Workshop on Managing Technical Debt (MTD). pp. 41–48, 2015.

[SP14] J. Yli-Huumo, A. Maglyas, K. Smolander, J. Haller and H. Törnroos. Developing Processes to Increase Technical Debt Visibility and Manageability - An Action Research Study in Industry. Product-Focused Software Process Improvement. pp. 368–378, 2016.

[SP15] Y. Guo, R. Oliveira Spínola, and C. Seaman. 2016. Exploring the costs of technical debt management - a case study. Empirical Softw. Engg. Volume 21(1), pp. 159–182, 2016.

[SP16] J. Yli-Huumo, A. Maglyas, and K. Smolander. How do software development teams manage technical debt? - An empirical study. Journal of System and Software, Vol. 120, C, pp. 195–218, 2016.

[SP17] U. Xiao, Yuanfang Cai, Rick Kazman, Ran Mo, and Qiong Feng. Identifying and quantifying architectural debt. 38th International Conference on Software Engineering (ICSE '16), pp. 488–498, 2016.

[SP18] S. Vidal, H. Vazquez, J. A. Diaz-Pace, C. Marcos, A. Garcia and W. Oizumi. JSpIRIT: a flexible tool for the analysis of code smells. 34th International Conference of the Chilean Computer Science Society (SCCC), pp. 1–6, 2015.

[SP19] A. Choudhary and P. Singh. Minimizing Refactoring Effort through Prioritization of Classes based on Historical, Architectural and Code Smell Information. QuASoQ/TDA@APSEC, 2016.

[SP20] R.K. Gupta, P. Manikreddy, S. Naik, and K. Arya. Pragmatic Approach for Managing Technical Debt in Legacy Software Project. 9th India Software Engineering Conference (ISEC '16), pp. 170–176, 2016.

[SP21] Z. Codabux and B. J. Williams. Technical debt prioritization using predictive analytics. 38th International Conference on Software Engineering Companion (ICSE '16), pp. 704–706, 2016.

[SP22] S. H. Vathsavayi and K. Systä. Technical Debt Management with Genetic Algorithms. 42th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), Limassol, pp. 50–53, 2016.

[SP23] S. Akbarinasaji, A. Bener and A. Neal. A Heuristic for Estimating the Impact of Lingering Defects: Can Debt Analogy Be Used as a Metric?. 8th Workshop on Emerging Trends in Software Metrics (WETSoM), pp. 36–42, 2017.

[SP24] L. Ferrera Ribeiro, N. S. R. Alves, M. G. d. M. Neto and R. O. Spínola. A Strategy Based on Multiple Decision Criteria to Support Technical Debt Management. 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA), pp. 334–341, 2017.

[SP25] Z. Codabux, B. Williams, G. Bradshaw and M. Cantor. An empirical assessment of technical debt practices in industry. Journal of Software: Evolution and Process. Vol. 29, 2017.

[SP26] S. Charalampidou, A. Ampatzoglou, A. Chatzigeorgiou, and P. Avgeriou. Assessing code smell interest probability: a case study. XP2017 Scientific Workshops (XP '17), Article 5, 8 pages, 2017.

[SP27] T. Besker, A. Martini and J. Bosch. Impact of Architectural Technical Debt on Daily Software Development Work — A Survey of Software Practitioners. 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA), pp. 278–287, 2017.

[SP28] M. Farias, J. Amâncio Santos, M. Kalinowski, M. Mendonça and R. Spínola, Rodrigo. Investigating the Identification of Technical Debt Through Code Comment Analysis. Lecture Notes in Business Information Processing. pp. 284–309, 2017.

[SP29] M. Ciolkowsk, L. Guzmán, A. Trendowicz and F. Salfner. Lessons Learned from the ProDebt Research Project on Planning Technical Debt Strategically. International Conference on Product-Focused Software Process Improvement. pp. 523–534, 2017.

[SP30] H. Ghanbari, T. Besker, A. Martini and J. Bosch. Looking for Peace of Mind? Manage Your (Technical) Debt: An Exploratory Field Study. International Symposium on Empirical Software Engineering and Measurement (ESEM), pp. 384–393, 2017.

[SP31] A. Martini and J. Bosch. Revealing Social Debt with the CAFFEA Framework: An Antidote to Architectural Debt. International Conference on Software Architecture Workshops (ICSAW),pp. 179–181, 2017

[SP32] A. Martini, S. Vajda, J. Vasa, A. Jones, M. Abdelrazek, J. Grundy and J. Bosch. Technical debt interest assessment: from issues to project. XP2017 Scientific Workshops. pp. 1–6, 2017.

[SP33] A. Martini and J. Bosch. The magnificent seven: towards a systematic estimation of technical debt interest. IXP2017 Scientific Workshops (XP '17), Article 7, 5 pages, 2017.

[SP34] T. Besker, A. Martini and J. Bosch. The Pricey Bill of Technical Debt: When and by Whom will it be Paid?. International Conference on Software Maintenance and Evolution (ICSME), pp. 13–23, 2017

[SP35] A. Martini, E. Sikander, and N. Madlani. A semi-automated framework for the identification and estimation of Architectural Technical Debt. Information and Software Technology, Vol. 93, C, pp. 264–279, 2018.

[SP36] J. M. Conejero, R. Rodríguez-Echeverría, J. Hernández, P. J. Clemente, C. Ortiz-Caraballo, E. Jurado, F. Sánchez-Figueroa, Early evaluation of technical debt impact on maintainability. Journal of Systems and Software. Vol. 142, pp. 92–114, 2018.

[SP37] A. Martini, T. Besker, J. Bosch. Technical Debt tracking: Current state of practice: A survey and multiple case study in 15 large organizations. Science of Computer Programming. Vol. 163, pp. 42–61, 2018.

[SP38] A. Martini, F. Arcelli Fontana, A. Biaggi, R. Roveda. Identifying and Prioritizing Architectural Debt Through Architectural Smells: A Case Study in a Large Software Company. 12th European Conference on Software Architecture (ECSA), pp. 24–28, 2018

[SP39] A. Tornhill. Prioritize technical debt in large-scale systems using codescene. International Conference on Technical Debt (TechDebt '18), pp. 59–60, 2018

[SP40] M. Albarak and R. Bahsoon. Prioritizing technical debt in database normalization using portfolio theory and data quality metrics. International Conference on Technical Debt (TechDebt '18), pp. 31–40, 2018

[SP41] H.M. Firdaus and H. Lichter. Towards a Technical Debt Management Framework based on Cost-Benefit Analysis. ICSEA 2015, 2015

[SP42] R. Plösch, J. Bräuer, M. Saft, and C. Körner. Design debt prioritization: a design best practice-based approach. International Conference on Technical Debt (TechDebt '18), pp. 95–104, 2018

[SP43] R. Rebouças de Almeida, U. Kulesza, C. Treude, D. Cavalcanti Feitosa and A. Higino Guedes Lima. Aligning Technical Debt Prioritization with Business Objectives: A Multiple-Case Study. International Conference on Software Maintenance and Evolution (ICSME 2018), pp. 655–664, 2018

[SP44] R. Alfayez and B. Boehm. Technical Debt Prioritization: A Search-Based Approach. 19th International Conference on Software Quality, Reliability and Security (QRS), pp. 434–445, 2019

References

Alfayez, R., Alwehaibi, W., Winn, R., Venson, E., Boehm, B., 2020. A systematic literature review of technical debt prioritization. In: International Conference on Technical Debt 2020.

Alves, N.S., Mendes, T.S., de Mendonça, M.G., Spínola, R.O., Shull, F., Seaman, C., 2016. Identification and management of technical debt: A systematic mapping study. Inf. Softw. Technol. 70, 100–121.

Ampatzoglou, A., Ampatzoglou, A., Chatzigeorgiou, A., Avgeriou, P., 2015. The financial aspect of managing technical debt: A systematic literature review. Inf. Softw. Technol. 64, 52–73.

Arcelli Fontana, F., Pigazzini, I., Roveda, R., Zanoni, M., 2016. Automatic detection of instability architectural smells. In: Proc. 32nd Intern. Conf. on Software Maintenance and Evolution (ICSME 2016). IEEE, Raleigh, North Carolina, USA.

Avgeriou, P., Kruchten, P., Nord, R.L., Ozkaya, I., Seaman, C., 2016a. Reducing friction in software development. IEEE Softw. 33 (1), 66–73.

Avgeriou, P., Kruchten, P., Ozkaya, I., Seaman, C., 2016b. Managing technical debt in software engineering (dagstuhl seminar 16162). Dagstuhl Rep. 6 (4), 110–138.

Avgeriou, P.C., Taibi, D., Ampatzoglou, A., Arcelli Fontana, F., Besker, T., Chatzigeorgiou, A., Lenarduzzi, V., Martini, A., Moschou, N., Pigazzini, I., Saarimaki, N., Sas, D.D., de Toledo, S.S., Tsintzira, A.A., 2020. An overview and comparison of technical debt measurement tools. IEEE Software 0-0.

Behutiye, W.N., Rodríguez, P., Oivo, M., Tosun, A., 2017. Analyzing the concept of technical debt in the context of agile software development: A systematic literature review. Inf. Softw. Technol. 82, 139–158.

Besker, T., Martini, A., Bosch, J., 2018a. Managing architectural technical debt: A unified model and systematic literature review. J. Syst. Softw. 135, 1–16.

Besker, T., Martini, A., Bosch, J., 2018b. Technical debt cripples software developer productivity: A longitudinal study on developers' daily software development work. In: International Conference on Technical Debt. In: TechDebt '18, pp. 105–114.

Besker, T., Martini, A., Bosch, J., 2019. Technical debt triage in backlog management. In: International Conference on Technical Debt. In: TechDebt '19, IEEE Press, Piscataway, NJ, USA, pp. 13–22.

Besker, T., Martini, A., Lokuge, R.E., Blincoe, K., Bosch, J., 2018c. Embracing technical debt, from a startup company perspective. In: 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME). pp. 415–425.

Braun, V., Clarke, V., 2006. Using thematic analysis in psychology. Qual. Res. Psychol. 3, 77–101.

Chatzigeorgiou, A., Ampatzoglou, A., Ampatzoglou, A., Amanatidis, T., 2015. Estimating the breaking point for technical debt. In: International Workshop on Managing Technical Debt (MTD). pp. 53–56.

Cunningham, W., 1992. The wycash portfolio management system. SIGPLAN OOPS Mess. 4 (2), 29–30.

Dybå, T., Dingsøyr, T., 2008. Empirical studies of agile software development: A systematic review. Inf. Softw. Technol. 50 (9–10), 833–859.

Fernández-Sánchez, C., Garbajosa, J., Yage, A., Perez, J., 2017. Identification and analysis of the elements required to manage technical debt by means of a systematic mapping study. J. Syst. Softw. 124, 22–38.

Fontana, F.A., Pigazzini, I., Roveda, R., Tamburri, D., Zanoni, M., Nitto, E.D., 2017. Arcan: A tool for architectural smells detection. In: International Conference on Software Architecture Workshops (ICSAW). pp. 282–285.

Khomyakov, I., Makhmutov, Z., Mirgalimova, R., Sillitti, A., 2019. Automated measurement of technical debt: A systematic literature review. In: ICEIS.

Kitchenham, B., Brereton, P., 2013. A systematic review of systematic review process research in software engineering. Inf. Softw. Technol. 55 (12), 2049–2075.

Kitchenham, B., Charters, S., 2007. Guidelines for performing systematic literature reviews in software engineering.

Li, Z., Avgeriou, P., Liang, P., 2015. A systematic mapping study on technical debt and its management. J. Syst. Softw. 101, 193–220.

Martini, A., 2018. Anacondebt: A tool to assess and track technical debt. In: International Conference on Technical Debt. In: TechDebt '18, pp. 55–56.

Martini, A., Bosch, J., 2016a. A multiple case study of continuous architecting in large agile companies: Current gaps and the CAFFEA framework. In: Conference on Software Architecture (WICSA). pp. 1–10.

Martini, A., Bosch, J., 2016b. An empirically developed method to aid decisions on architectural technical debt refactoring: Anacondebt. In: International Conference on Software Engineering Companion (ICSE-C). pp. 31–40.

Martini, A., Bosch, J., 2016c. An empirically developed method to aid decisions on architectural technical debt refactoring: AnaConDebt. In: Proceedings of the 38th International Conference on Software Engineering Companion, ICSE '16. pp. 31–40.

Martini, A., Bosch, J., Chaudron, M., 2015. Investigating architectural technical debt accumulation and refactoring over time: A multiple-case study. Inf. Softw. Technol. 67, 237–253.

McConnell, S., 2013. Managing technical debt. http://www.sei.cmu.edu/community/td2013/program/upload/techncaldebt-icse.pdf.

Ribeiro, L.F., Farias, M.A.d.F., Mendonça, M., Spínola, R.O., 2016. Decision criteria for the payment of technical debt in software projects: A systematic mapping study. In: 18th International Conference on Enterprise Information Systems, ICEIS 2016, Portugal, pp. 572–579.

Rios, N., de Mendonça Neto, M.G., Spínola, R.O., 2018. A tertiary study on technical debt: Types, management strategies, research trends, and base information for practitioners. Inf. Softw. Technol. 102, 117–145.

Seaman, C.B., Guo, Y., Izurieta, C., Cai, Y., Zazworka, N., Shull, F., Vetro, A., 2012. Using technical debt data in decision making: Potential decision approaches. In: 2012 Third International Workshop on Managing Technical Debt (MTD). pp. 45–48.

Tom, E., Aurum, A., Vidgen, R., 2013. An exploration of technical debt. J. Syst. Softw. 86 (6), 1498–1516.

Wohlin, C., 2014. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In: EASE 2014.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., 2012. Experimentation in Software Engineering. Springer.

Valentina Lenarduzzi is a postdoctoral researcher at the LUT University in Finland. Her primary research interest is related to data analysis in software engineering, software quality, software maintenance and evolution, with a special focus on Technical Debt. She obtained her Ph.D. in Computer Science at the Università degli Studi dell'Insubria, Italy, in 2015, working on data analysis in Software Engineering. She also spent 8 months as Visiting Researcher at the Technical University of Kaiserslautern and Fraunhofer Institute for Experimental Software Engineering (IESE) working on Empirical Software Engineering in Embedded Software and Agile projects. In 2011 she was one of the co-founders of Opensoftengineering s.r.l., a spinoff company of the Università degli Studi dell'Insubria. Contact her valentina.lenarduzzi@lut.fi.

Terese Besker is a Ph.D. candidate in the Software Engineering at Chalmers University of Technology in Sweden. She is working in the research fields of technical debt management. Before becoming a Ph.D. student, she had worked as a senior software engineer in the software industry for more than fifteen years. She also has a bachelor's degree in software engineering and a master's degree in applied IT. She has published several peer-reviewed articles in journals, conference and workshop proceedings. Contact her at besker@chalmers.se.

Antonio Martini is Associate Professor at the University of Oslo and is a part-time researcher at Chalmers University of Technology. The current focus of Antonio's research is on Technical Debt, Architecture, Technical Leadership and Agile software development. Antonio's experience covers Software Engineering and Management in several contexts: large, embedded software companies, small, web companies, business to business companies, startups. HIs expertise ranges from technical programming to software architecture and software quality, to Agile ways of working and software business. Antonio Martini has worked as Principal Strategic Researcher at CA Technologies for a co-financed project for technology transfer related to Technical Debt and Architecture by the H2020 Marie Skłodowska-Curie grant of the European Union. Antonio has collaborated with several large companies such as Ericsson, Volvo, Saab, Axis, Grundfos, Siemens, Bosch and Jeppesen. He has also started his own consultancy company and have run projects with large companies in north- and central-Europe to manage and visualize Technical Debt. Antonio has been employed as a Postdoc Researchers at Chalmers, after having obtained a Ph.D. in Software Engineering at Chalmers University of Technology, Sweden in 2015. Contact him at antonima@ifi.uio.no.

Davide Taibi is an associate professor (tenure track) at the Tampere University, Finland. He obtained his Ph.D. in Computer Science at the Università degli Studi dell'Insubria, Italy in 2011. His research activities are focused on software quality in cloud-based systems, supporting companies in keeping Technical Debt under control while migrating to cloud-native architectures. Moreover, he is interested in patterns, anti-patterns and "bad smells" that can help companies to avoid issue during the development process both in monolithic systems and in cloud-native ones. Formerly, he worked at the Free University of Bolzano, Technical University of Kaiserslautern, Germany, Fraunhofer IESE - Kaiserslautern, Germany, and Università degli Studi dell'Insubria, Italy. In 2011 she was one of the co-founders of Opensoftengineering s.r.l., a spin-off company of the Università degli Studi dell'Insubria. Contact him at davide.taibi@tuni.fi.

Francesca Arcelli Fontana is an of Full Professor at University of Milano Bicocca. She had her Master degree and Ph.D. in Computer Science taken at the University of Milano. The actual research activity principally concerns the software engineering field. In particular in software evolution, reverse engineering, managing technical debt, and software quality assessment. She is at the head of the Software Evolution and Reverse Engineering Lab at University of Milano Bicocca. Contact her at francesca.arcelli@unimib.it.